UIMA Asynchronous Scaleout

Release 0.5

This edition applies to the IBM Unstructured information Management Architecture Asynchronous Scaleout Edition (UIMA AS) Version 0.5.0 and to all subsequent release and modifications until otherwise indicated in new editions.

Published August, 2007

Table of Contents

Chapter 1. Overview - Asynchronous Scaleout

UIMA Asynchronous Scaleout (AS) is a set of capabilities supported in the UIMA Framework for achieving scaleout that is more general than the approaches provided for in the Collection Processing Manager (CPM). AS is a second generation design, replacing the CPM and Vinci Services. The CPM and Vinci are still available and are not being deprecated, but new designs are encouraged to use AS for scalability, and current designs reaching limitations want to move to AS.

AS is integrated with the new flow controller architecture, and can be applied to both primitive and aggregate analysis engines.

1.1. Terminology

Terms used in describing AS capabilities include:

AS
Asynchronous Scaleout - a name given to the capability described here

AS-JMS/AMQ/Spring
A variety of AS, based on JMS (Java Messaging Services), Active MQ, an Apache Open Source implementation of JMS, and the Spring framework. This variety is the one described in detail in this document.

Queue
Queues are the basic mechanism of asynchronous communication. One or more "producers" send messages to a queue, and a queue can have one or more "consumers" that receive messages. Messages in UIMA AS are usually CASes, or references to CASes. Queues are identified by a 2 part name. The first part is the Queue Broker; the second part is a Queue Name.

AS Component
An AS client or service. AS clients send requests to AS service queues and receive back responses on reply queues. AS services can be AS Primitives or AS aggregates (see following).

AS Primitive
An AS service that is either a Primitive Analysis Engine or an Aggregate AE whose Delegates are **not** AS-enabled

AS Aggregate
An AS service that is an Aggregate Analysis Engine where the Delegates are also AS components.

AS Client
A component sending requests to AS services. An AS client is typically an application using the UIMA AS client API, a JMS Service Client Proxy, or an AS Aggregate.

co-located

two running pieces of code are co-located if they run in the same JVM and share the same UIMA framework implementation and components.

Queue Broker

Queue brokers manage one or more named queues. The brokers are identified using a URL, representing where they are on the network. When the queue broker is co-located with the AS client and service, CASes are passed by reference, avoiding serialization / deserialization.

Transport Connector

AS components connect to queue brokers via transport connectors. UIMA AS will typically use "tcp" connectors. "http" connectors are useful for tunneling via an existing public web server.

1.2. AS versus CPM

It is useful to compare and contrast the approaches and capabilities of AS and CPM.

	AS	CPM
Putting components together	Aggregates are the only way to put components together.	**Two methods of putting components together** 1. CPE (Collection Processing Engine) descriptor, which has sections specifying a Collection Reader, and a set of CAS Processors 2. Each CAS Processor can, as well, be an aggregate
Kinds of Aggregates	An aggregate can be run **asynchronously** using the AS mechanism, with a queue in front of each delegate, or it can by run **synchronously**. When run asynchronously, *all* of the delegates will have queues in front of them, and AS Primitive delegates can be individually scaled out (replicated) as needed.	All aggregates are run synchronously. In an aggregate, only one component is running at a time.
CAS flow	Any, including custom user-defined sequence using user-provided flow controller.	Fixed linear flow between CAS processors. A single CAS processor can be an aggregate, and within the aggregate, can have any flow including custom user-defined sequence using user-provided flow controller.

	AS	CPM
Threading	Each instance of a component runs in its own thread.	One thread for the collection reader, one for the CAS Consumers, "n" threads for the main pipeline.
Delegate deployment	Co-located or remote.	Co-located or remote.
Life cycle management	Scripts to launch services, launch Queue Brokers.	Scripts to launch services, start Vinci Name Service. In addition, CPE "managed" configuration provides for automatic launching of UIMA Vinci services in same machine, in different processes.
Error recovery	Similar capabilities as the CPM provides for CAS Processors, but at the finer granularity of each AS component. The support includes customizable behavior overrides and extensions via user code.	Error detection, thresholding, and recovery options at the granularity of CAS Processors (which are CPM components, not delegates of aggregates), with some customizable callback notifications
Firewall interactions	Enables deployment of AS services behind a firewall using a public broker. Enables deployment of a public broker through single port, or using HTTP "tunneling".	When using Vinci protocol, requires opening a large number of ports for each deployed service. SOAP connected services require one open port.
Monitoring	JMX (Java Management Extensions) are enabled for recording many kinds of statistical information, and can be used to monitor (and, in the future, control) the operations of AS configured systems.	Limited JMX information
Collection Reader	Supported for backwards compatibility. New programs should use the CAS Multiplier instead, which is more general, or have the application pass in CASes to be processed. The compatibility support wraps Collection Readers as Cas Multipliers.	Is always first element in linear CPE sequence chain

1.3. Design goals for Asynchronous Scaleout

The design goals for AS are:

1. Increased flexibility and options for scaleout (versus CPM)
    a. scale out parts independently of other parts, to appropriate degree
    b. more options for protocols for remote connections, including some that don't require many ports through firewalls
2. Build upon widely accepted Apache-licensed open source middleware
3. Simplification:
    a. Standardize on single approach to aggregate components
    b. More uniform Error handling / recovery / monitoring for all AS managed components.

1.4. AS Concepts

1.4.1. User written components and multi-threading

AS provides for scaling out of annotators - both aggregates and primitives. Each of these can specify a user-written implementation class. For primitives, this is the annotator class with the process() method that does the work. For aggregates, this can be an (optional) custom flow controller class that computes the flow.

The classes for annotators and flow controllers do not need to be "thread-safe" with respect to their instance data - meaning, they do not need to be implemented with synchronization locks for access to their instance data, because each instance will only be called using one thread at a time. Scale out for these classes is done using multiple instances of the class.

However, if you have class "static" fields shared by all instances, or other kinds of external data shared by all instances (such as a writable file), you must be aware of the possibility of multiple threads accessing these fields or external resources, running on separate instances of the class, and do any required synchronization for these.

1.4.2. AS Component wrapping

Components managed by AS

1. have an associated input queue (this may be internal, or explicit and externalized.

    They receive work units (CASes) from this queue, and return the updated CASes to an output queue which is specified as part of the message delivering the input work unit (CAS).

2. have a container which wraps the component and provides the following services (see Figure 1.1, “AS Primitive Wrapper” [5]):
    - A connection to an input queue of CASes to be processed
    - Scale-out within the JVM for components at the bottom level - the AS Primitives. Scaleout creates multiple instances of the annotator(s), and runs each one on its own thread, all drawing work from the same input queue.

- (For AS Aggregates) connections to input queues of the delegates
- A "pull" mechanism for the component to pull new CASes (to be processed) from their associated input queue
- (For AS Aggregates) A separate, built-in internal queue to receive CASes back from delegates. These are passed to the aggregate's flow controller, which then specifies where they go next.
- A connection to user-specified error handlers. Error conditions are communicated to the flow controller, to enable user / dynamically determined recovery or termination actions.

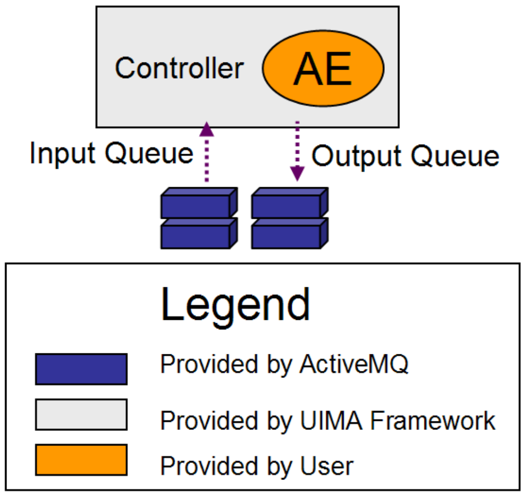

Figure 1.1. AS Primitive Wrapper

As shown in the next figure, when the component being wrapped is an AS Aggregate, the container will use the aggregate's flow controller (shown as "FC") to determine the flow of the CASes among the delegates. The next figure shows the additional output queue configured for aggregates to receive CASes returning from delegates. The dashed lines show how the queues are associated with the components.

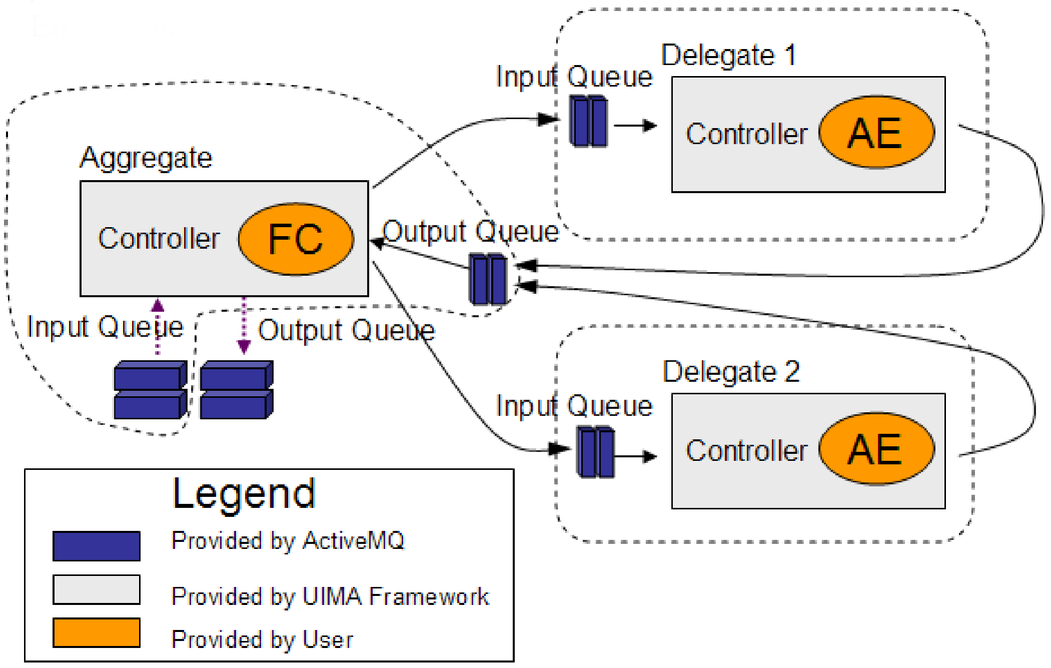

Figure 1.2. AS Aggregate wrapper

The collection of parts and queues is wired together according to a deployment specification, provided by the deployer. This specification is a collection of one or more deployment descriptors.

1.4.3. Deployment alternatives

Deployment is concerned with the following kinds of parts, and allocating these parts (possibly replicated) to various hosts:

- Application Drivers. These represent the top level caller of UIMA functionality. Examples include: stand-alone Java applications, such as the example document analyzer tool, a custom Web servlet, etc.
- AS Services. AS primitive or AS aggregate services deployed on one or more nodes as needed to meet scalability requirements.
- Queue Brokers. Each Queue Broker manages and provides the storage facility for one or more named queues.

Parts can be co-located or not; when they're not, we say they're remote. Remote includes running on the same host, but in a different process space, using a different JVM or other native process. Connections between the parts are done using the JMS (Java Messaging Service) protocols, and in this version the support for this is provided by the ActiveMQ implementation from Apache.org.

Note: For high availability, the Queue Brokers can be, themselves, replicated over many hosts, with fail-over capability provided by the underlying ActiveMQ implementation.

1.4.3.1. Configuring multiple instances of components

AS components can be replicated; the replicated components can be co-located or distributed across different nodes. The purpose of the replication is to allow multiple work units (CASes) to be processed in parallel, in multiple threads, either in the same host, or using different hosts. The vision is that the deployment is able to replicate just those components which are the bottleneck in overall system thruput.

There are two ways replication can be specified.

1. In the deployment descriptor, set the numberOfInstances attribute to a number bigger than one.

2. Deploy the same service on many nodes, specifying the same input service queue

The first way is limited to replicating an AS Primitive. An AS Primitive can be the whole component of the service, or it can be at the bottom of an aggregate hierarchy of co-located parts.

Replicating an AS Primitive has the effect of replicating all of its components, since no queues are used below its input queue.

1.4.3.2. Queues

Asynchronous operation is enabled by use queues to connect components. Each queue is defined by a queue name and the URL of its Queue Broker. AS services register as queue consumers to obtain CASes to work on (as input) and to send CASes they're finished with (as output) to the reply queue specified by the AS client.

For the AS-JMS/AMQ/Spring implementation, the queue implementation is provided by ActiveMQ queue broker. A single Queue Broker manages multiple queues. By default UIMA AS configures the Queue Broker to use in-memory queues, so the queue is resident on the same JVM as its managing Queue Broker. ActiveMQ offers serveral failsafe options, including the use of disc-based queues and redundant master/slave broker configurations.

The decisions about where to deploy Queue Brokers are deployment decisions, made based on issues such as domain of control, firewalls, CPU / memory resources, etc. Of particular interest for distributed applications is that a UIMA AS service can be deployed behind a firewall but still be publicly available by using a queue broker that is available publicly.

When components are co-located, an optimization is done so that CASes are not actually sent as they would be over the network; rather, a reference to the in-memory Java object is passed using the queue.

Warning: Do not hook up different kinds of services to the same input queue. The framework expects that multiple services all listening to a particular input queue are sharing the workload of processing CASes sent to that queue. The framework does not currently verify that all services on a queue are the same kind, but likely will in a future release.

1.4.3.3. Deployment Descriptors

Each deployment descriptor specifies deployment information for one service, including all of its co-located delegates (if any). A service is an AS component, having one top level input queue, to which CASes are sent for processing.

Each deployment descriptor has a reference to an associated Analysis Engine descriptor, which can be an aggregate, primitive (including CAS Consumers), or service client descriptor.

AS Components and their associated queue brokers can be co-located on the same host/jvm; the deployment descriptor indicates which components (if any) are co-located on its host/jvm, and specifies the remote queues (queue-brokers and queue-names) for all other components.

All services need to be manually started using an appropriate deployment descriptor (describing the things to be set up on that server).

Deploying UIMA aggregates

UIMA aggregates can either be run asynchronously as AS Aggregates, or synchronously (as AS Primitives). AS Aggregates have a queue in front of each delegate; results from each delegate are sent to a receiving (internal) queue. UIMA aggregates run as AS Primitives send CASes synchronously to each delegate, without using any queuing mechanism.

Each delegate in an AS Aggregate can be specified to be local or remote. Local means co-located using internal (hidden) queues; remote means all others, including delegates running in a different JVM, or in the same JVM but that can be shared by multiple clients For each delegate which is remote, the deployment descriptor specifies the delegate's input queue. If the delegate is local, an hidden, internal queue is automatically created for that delegate.

1.4.4. First implementation - Design limitations

This section describes limitations of the initial support for AS. Some of these limitations are due to the functionality being staged over several releases.

1.4.4.1. Sofa Mapping limits

Sofa mapping works for co-located delegates, only. As with Vinci and SOAP, remote delegates needing sofa mapping need to respecify sofa mappings in an aggregate descriptor at the remote node.

1.4.4.2. Parameter Overriding limits

Parameter overrides only work for co-located delegates. As with Vinci and SOAP, remote delegates needing parameter overrides need to respecify the overrides in an aggregate descriptor at the remote node.

1.4.4.3. Resource Sharing limits

Resource Sharing works for co-located delegates, only.

1.4.5. Compatibility with earlier version of remoting and scaleout

A Vinci client service descriptor can be used in an aggregate descriptor as before, and can be used as a primitive analysis engine in a deployment descriptor. There is a new type of client service descriptor for an AS service, the JMS service descriptor; see Section 1.7, “JMS Service Descriptor” [11]

1.5. Application Concepts

When UIMA is used, it is called using Application APIs. A typical top-level driver has this basic flow:

1. Read UIMA descriptors and instantiate components
2. Do a Run
3. Do another Run, etc.
4. Stop

Note: The initial release limits this flow to one run.

A "run", in turn, consists of 3 parts:

1. initialize (or reinitialize, if already run)
2. process CASes
3. finish (collectionProcessComplete is called)

Initialize is called by the framework when the instance is created. The other methods need to be called by the driver. `collectionProcessComplete` should be called when the driver determines that it is finished sending input CASes for processing using the `process()` method. `reinitialize()` can be called if needed, after changing parameter settings, to get the co-located components to reinitialize.

1.5.1. Application API

Please see the sample code.

1.5.2. Collection Process Complete

Applications may want to signal a chain of annotators being used in a particular "run" when all CASes for this run have been processed, and any final computation and outputting is to be done; it calls the collectionProcessComplete method to do this. This is frequently done when using statefull components which are accumulating information over multiple documents.

It is up to the application to determine when the run is finished and there are no more CASes to process. It then calls this method on the top level analysis engine; the framework propagates this method call to all delegates of this aggregate, and this is repeated recursively for all delegate aggregates.

This call is synchronous, meaning when this call is issued by an application, the framework will block the thread issuing the call until all processing of CASes within the aggregate has completed and the collectionProcessComplete method has returned (or timed out) from every component it was sent to.

Components receive this call in a fixed order taken from the <fixedFlow> sequence information in the descriptors, if that is available, and in an arbitrary order otherwise.

If a component is replicated, only one of the instances will receive the collectionProcessComplete call.

1.6. Monitoring and Controlling an AS application

JMX (Java Management Extensions) are used for monitoring and controlling an AS application. This capability is being staged; initial versions have some monitoring capability, but little controlling capability.

The first versions of AS will use the standard GUI tooling available as part of Java 5 to display the JMX results. Later versions may include additional UIMA-specific tooling for this.

1.6.1. Instrumentation provided

The implementation provides the following kinds of instrumentation via JMX:

- Timing
    - by component, by CAS(?)
    - by queue
    - message transit & serialization/deserialization
- component / host status
    - by component
    - state: OK, Idle, Working, Stopped, restarting, etc.

1.7. JMS Service Descriptor

To call a UIMA AS Service from Document Analyzer or any other UIMA "SE" application, use a descriptor such as the following:

```
<customResourceSpecifier xmlns="http://uima.apache.org/resourceSpecifier">
   <resourceClassName>
      com.ibm.uima.aae.jms_adapter.JmsAnalysisEngineServiceAdapter
   </resourceClassName>
   <parameters>
     <parameter name="brokerURL"
       value="tcp://uima17.watson.ibm.com:61616"/>
     <parameter name="endpoint"
       value="uima.ee.RoomDateMeetingDetectorAggregateQueue"/>
   </parameters>
</customResourceSpecifier>
```

The resouceClassName must be set exactly as shown. Set the brokerURL and endpoint parameters to the appropriate values for the UIMA AS Service you want to call. These are the same settings you would use in a deployment descriptor to specify the location of a remote delegate. Note that this is a synchronous adapter, which processes one CAS at a time, so it will not take advantage of the scalability that UIMA AS provides. To process more than one CAS at a time, you must use the Asynchronous UIMA AS Client.

For more information on the customResourceSpecifier see Section 2.8, “Custom Resource Specifiers” in *UIMA References*.

1.8. Collection Reader support

Collection Readers are supported for backwards compatibility; new programs should use the Cas Multiplier. The compatibility is achieved by wrapping the Collection Reader so that it looks like a Cas Multiplier. Because of this implementation, you can use a CollectionReader descriptor anywhere that a CAS Multiplier descriptor would work. Calls to the CAS Multiplier's next() method are translated into calls to the Collection Reader's getNext() method. Since a Collection Reader cannot accept a CAS as input, calls to the CAS Multiplier's process(CAS) method will be translated into calls to the Collection Reader's reconfigure() method (except for the very first call to process(), which is ignored). This is done so that if a Collection Reader reacts to reconfigure() by resetting its state to be at the beginning of the collection, then when deployed as a CAS Multiplier service it can be reused multiple times without having to restart the service.

Chapter 2. Error Handling for Asynchronous Scaleout

This chapter discusses the high level architecture for Error Handling from the user's point of view.

2.1. Basic concepts

This chapter describes error configuration for AS components. The service client descriptor for connecting to an AS component also has some configuration related to errors, but that is discussed elsewhere.

The AS framework manages a collection of component parts written by users (user code) which can throw exceptions. In addition, the AS framework can run timers when sending commands to user code which can create timeouts.

An AS component is either an AS aggregate or an AS primitive. AS aggregates can have multiple levels of aggregation; error configuration is done for each level of aggregation. The rest of this chapter focuses on the error configuration one level at a time (either for one particular level in an aggregate hierarchy, or for an AS primitive).

There is a small number of commands which can be sent to an AS component. When a component returns the result, if an error occurs, an error result is returned instead of the normal result.

Configuration and support is provided for three classes of errors:

1. Exceptions thrown from code (component or framework) at this level
2. error messages received from delegates.
3. timeouts of commands sent to delegates.

The second and third class of errors is only possible for AS aggregates.

When errors happen, the framework provides a standard set of configurable actions. These can be extended with (optional) user-supplied error handling code. See Section 2.8, “Commands and allowed actions” [18] for a summary table of the actions available in different situations.

2.2. Associating Errors with incoming commands

Components managed by AS may generate errors when they are sent a command. The error is associated with the command that was running to produce the error.

There are three incoming message commands supported by the AS framework:

1. getMetadata - sent by the framework when setting up a new run

2. processCas - sent repeatedly, once for each CAS
3. collectionProcessComplete - sent when an application calls this method

Error handling actions are associated with these various commands. Some error handling actions make sense only if there is an associated CAS object, and are therefore only allowed with the processCas command.

2.3. Error handling overview

When an error happens, it is either "recovered", or not; only errors from delegates of an AS aggregate can be recovered.

Commands normally return results; however if an non-recoverable error occurs, the command returns an error result instead.

For AS aggregates, each level in aggregate hierarchy can be configured to try to recover the error. If a particular AS aggregate level does not recover, the error is sent up to the next level of the hierarchy (or to the calling client, if a top level). The error result is updated to reflect the actions the framework takes for this error.

Non-recovered errors can optionally have an associated "Terminate" or "Disable" action (see below), triggered by the error when a threshold is reached.

These actions, if they occur, are recorded in the error result, indicating that the additional Terminate or Disable action occurred.

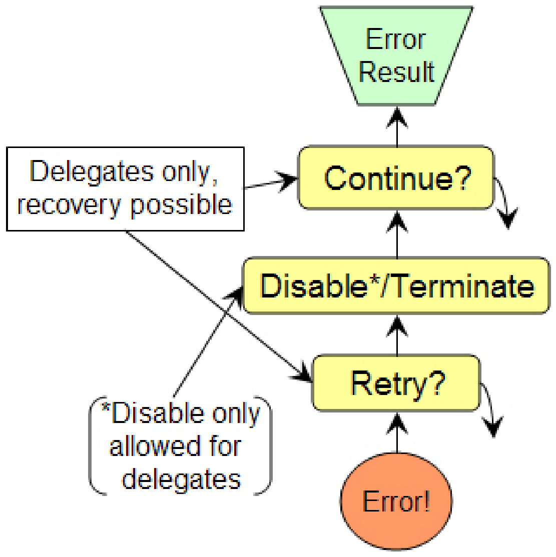
Error Result
Delegates only, recovery possible
Continue?
Disable*/Terminate
Retry?
*Disable only allowed for delegates
Error!

Figure 2.1. Basic error handling chain for AS Aggregates for errors from delegates

The basic error handling chain starts with an error, and can attempt to recover using retry. If this fails (or is not configured), any Terminate or Disable action that is specified is examined to see if this error exceeded their threshold, and if so, the framework disables a delegate or terminates the entire component. If not, the error is counted for those thresholds, and recovery by the Continue action can be attempted. If that fails, an error result is returned to the caller.

2.4. Error results

When an error occurs and is not recovered, an error result is returned to the caller. This is an object with the following methods:

Table 2.1. Error Result

Method	**Description**
Throwable getRootCause()	Returns the underlying root cause first reported as an error
ErrorResultComponentPath getComponentKeyPath()	Returns a path consisting of a list of component key names (for delegates) or brokerURL + endpoint elements, for top-level components not contained in an aggregate, leading down to the component that first reported the error. The path starts with the current component, and ends with the component associated with the original error.
wasTerminated()	true if any termination occurred with this error
wasDisabled()	true if any disabling occurred with this error
ErrorResultTDs getTDs()	Returns a collection of path(s) to the component(s) that was/were terminated or disabled, together with a flag indicating which (could be both).

2.5. Error Recovery actions

When errors occur in delegates, the aggregate containing them can attempt to recover. There are two basic recovery actions: retrying the same command and continuing past (skipping) the failing component.

Every command sent to a delegate can have an associated (configurable) timeout. If the timeout occurs before the delegate responds, the framework creates an error representing the timeout.

> **Note:** If, subsequently, a response is (finally) received corresponding to the command that had timed-out, this is logged, but the response is discarded and no further action is taken.

When errors occur in delegates, retry is attempted (if configured), some number of times. If that fails, error counts are incremented and thresholds examined for Terminate/Disable actions. If those actions are not taken, Continue is attempted (if configured). If Continue fails, the error is not recovered, and the aggregate returns the error result from the delegate to the aggregate's caller.

2.5.1. Aggregate Error Actions

This section describes the error actions in more detail.

2.5.1.1. Retry

Retry is an action which re-sends the same command that failed back to the input queue of the delegate. (Note: It may be picked up by a different instance of that delegate, which may have a better chance of succeeding.)

Specifying Retry (or Continue) for the processCAS command sends the CAS to the delegate. If the delegate is remote, then the framework will keep a copy of the original CAS it previously sent, in order to have it to send if a retry occurs. Preserving the CAS before sending it to a remote delegate takes space and time; this must be considered when specifying retry or continue actions for a remote delegate.

If the delegate is co-located, the framework is sharing the CAS object with the delegate, and sends the same CAS to the delegate upon retry. In this case, the CAS may be partially updated by the delegate's previous call. If this is not OK, the designer should either not specify retry or continue recovery for this delegate, or change the delegate to be not co-located.

Co-located delegate **timeout errors** are not allowed to have retry or continue recovery, because this exposes the possibility of multiple threads operating on the same CAS at the same time.

Note that no CAS is required in order to retry the "getMetadata" command.

The "collectionProcessComplete" command is never retried.

Retry is done some number of times, as specified in the deployment descriptor.

2.5.1.2. Disable Action

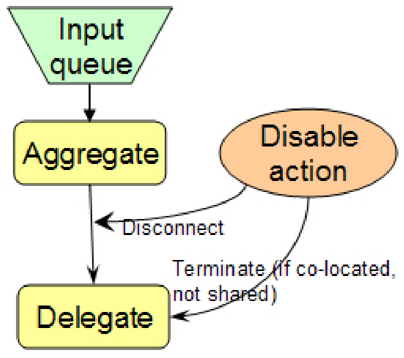

Figure 2.2. Disable action

The Disable action is allowed only on delegate errors. If specified, it marks the particular delegate causing the error as "disabled" so it is always skipped in subsequent calls. When this action is taken, the framework calls the flow controller, telling it to remove the particular delegate from the flow. If the flow controller cannot do this, it signals this to the framework, which converts this into a Terminate action, terminating the aggregate as a whole.

2.5.1.3. Continue Action

For processCas commands, the Continue action causes the framework to ask the flow controller to continue, if possible, by skipping the particular delegate for this CAS and going to the next one. If the flow controller decides it can't continue, perhaps because the failing delegate is critical to the operation of the aggregate, the framework returns the underlying error to the caller of the aggregate, with an indication that CAS processing was aborted. This also happens for errors if there is no Continue action.

For "collectionProcessComplete" commands, Continue means to ignore the error, and continue as if the collectionProcessComplete call had returned normally.

This action is not allowed for the getMetadata command.

2.6. Thresholds for Terminate and Disable

The Terminate and Disable actions are conditioned by testing against a threshold. Thresholds are specified as a "rate" based on processed CASes - so many errors per number of CASes processed.

Thresholds are specified as two numbers - an error count and a window. The threshold is exceeded if the number of errors occurring within the window size is equal to or greater than "error count". (A count of 0 for the error count is treated as a special value and disables the error threshold.)

Errors associated with the processCas command are the only ones that are counted in the threshold measurements.

2.7. Terminate Action

If the threshold is exceeded, the service represented by this component terminates, disconnecting itself as a listener from its input queue, and cleaning itself up (releasing resources, etc.). During cleanup, the component analysis engine's `destroy` method is called.

> The termination action is cascaded down to all co-located, non-shared delegates, if any.

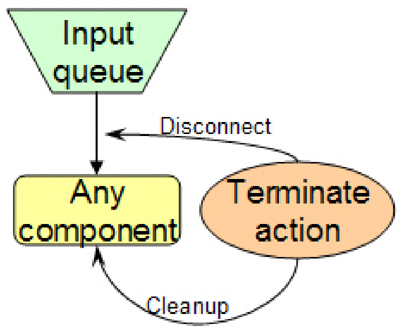

Figure 2.3. Terminate action

If the threshold is not exceeded, the error counts associated with the threshold are incremented.

Note: Some errors will always cause a terminate: for instance, framework or flow controller errors cause immediate termination.

2.8. Commands and allowed actions

All of the allowed actions are optional, and default to not being done, except for getMetadata being sent to a delegate that is remote - this has a default timeout of 1 minute.

Here's a table of the allowed actions, by command. In this table, the Retry, Continue, and Disable actions apply to the particular Delegate associated with the error; the Terminate action applies to the entire component.

The framework returns an Error Result to the caller for errors that are not recovered.

Table 2.2. Error actions by command type

Command	**Error actions allowed**	
	AS Aggregate	**AS Primitive**
getMetadata	Retry, Disable, Terminate	Terminate
processCas	Retry, Continue, Disable, Terminate	Terminate
collection Processing Complete	Continue, Disable, Terminate	Terminate

Chapter 3. Asynchronous Scaleout Deployment Descriptor

3.1. Descriptor Organization

Each deployment descriptor describes one service, associated with a single UIMA descriptor (aggregate or primitive), and describes the deployment of those UIMA components that are co-located, together with specifications of connections to those subcomponents that are remote.

The deployment descriptor is used to augment information contained in an analysis engine descriptor. It adds information concerning

- which components are managed using AS
- queue names for connecting components
- error thresholds and recovery / terminate action specifications
- error handling routine specifications

The application can include both Java and non-Java components; the deployment descriptors may be slightly different for non-Java components.

3.2. Deployment Descriptor

Each deployment descriptor describes components associated with one UIMA descriptor. The basic structure of a Deployment Descriptor is as follows:

```
<analysisEngineDeploymentDescription
      xmlns="http://uima.apache.org/resourceSpecifier">

  <!-- the standard (optional) header -->
  <name>[String]</name>
  <description>[String]</description>
  <version>[String]</version>
  <vendor>[String]</vendor>

  <deployment protocol="jms" provider="activemq">

    <casPool numberOfCASes="xxx" />

    <service>                           <!-- must have only 1 -->
      <inputQueue .../>

      <topDescriptor .../>
                                                 <!-- 0 or more -->
      <analysisEngine key="key name" async="[true/false]">

        <scaleout numberOfInstances="1"/>       <!-- optional  -->
        <casMultiplier poolSize="5"/>           <!-- optional  -->
        <asyncPrimitiveErrorConfiguration .../> <!-- optional  -->

        <delegates>      <!-- optional, only for aggregates -->
                                              <!-- 0 or more -->
          <analysisEngine key="key name" async="[true/false]">
                ...        <!-- optional nested specifications -->
          </analysisEngine>
                . . .
          <remoteAnalysisEngine key="key name"> <!-- 0 or more -->
            <casMultiplier poolSize="5"/>       <!-- req | omit-->
            <inputQueue ... />
            <replyQueue location="[local|remote]"/><!-- optional-->
            <serializer method="xmi"/>
            <asyncAggregateErrorConfiguration ... />
          </remoteAnalysisEngine>
                . . .
        </delegates>
      </analysisEngine>
    </service>
  </deployment>
</analysisEngineDeploymentDescription>
```

3.3. CAS Pool

This element specifies information for managing CAS pools. Having more CASes in the pools enables more AS components to run at the same time. For instance, if your application had four components, but one was slow, you might deploy 10 instances of the slow component. To get all 10 instances working on CASes simultaneously, your CAS pool should be at least 10 CASes. The casPool size should be small enough to avoid paging.

3.4. Service

This section is required and specifies the deployment information for the service.

3.5. Input Queue

The inputQueue element is required. It identifies the input queue for the service.

```
<inputQueue brokerURL="tcp://x.y.z:portnumber"
    endpoint="queue_name"
    prefetch="1"/>
```

The queue broker address includes a protocol specification, which should be set to either "tcp", or "http". The brokerURL attribute specifies the queue broker URL, typically its network address and port. .

The http protocol is similar to the tcp protocol, but is preferred for wide-area-network connections where there may be firewall issues, as it supports http tunnelling.

> **Warning:** When remote delegates are being used, and the replyQueue is remote, the brokerURL value used for this remote delegate is used also for the remote reply Queue, and must be valid for both the client to send requests and the remote service to send replies to. The URL to use for the reply is resolved on the remote system when sending a reply. Using "localhost" will not work, nor will partially specified URLs unless they resolve to the same URL on all nodes where services are running. The recommended best practice is to use fully qualified URL names.

The queue name is used to uniquely identify a queue belonging to a particular broker.

The `prefetch` attribute controls prefetching of messages for an instance of the service. It can be 0 - which disables prefetching. This is useful in some realtime applications for reducing latency. In this case, when a new request arrives, any available instance will take the request; if prefetching was set above 0, the request might be prefetched by a busy service. The default value if not specified is 1.

> **Note:** The `prefetch` attribute is only used with the top inputQueue element for the service.

3.6. Top level Analysis Engine descriptor

Each service must indicate some analysis engine to run, using this element.

```
<topDescriptor>
  <import location="..." /> <!-- or name="..." -->
</topDescriptor>
```

This is the standard UIMA import element. Imports can be by name or by location; see Section 2.2, “Imports” in *UIMA References*.

3.7. Analysis Engine

This is used to describe an element which is an analysis engine. It is optional and only needed if the defaults are being overridden. The `async` attribute is only used for aggregates, and specifies that this aggregate will be run asynchronously (with input queues in front of all of its delegates) or not. If not specified, the async property defaults to "false" except in the case where the deployment descriptor includes the <delegates> element, when it defaults to "true". If you specify async="false", then it is an error to specify any <delegates> in the deployment descriptor.

The `key` attribute must have as its value the key name used in the containing aggregate descriptor to uniquely identify this delegate. Since the top level aggregate is not contained in another aggregate, this can be omitted for that element. Deployment information is matched to delegates using the key name specified in the aggregate descriptor to identify the delegate.

```
<analysisEngine key="key name" async="true">
```

```
  <scaleout numberOfInstances="1"/>        <!-- optional  -->
  <casMultiplier poolSize="5"/>            <!-- req | omit-->

    <!-- next two are optional, but only one allowed -->
  <asyncAggregateErrorConfiguration .../>  <!-- optional  -->
  <asyncPrimitiveErrorConfiguration .../>  <!-- optional  -->

  <delegates>                              <!-- optional  -->
    <analysisEngine key="key name" ...>    <!-- 0 or more -->
            ...       <!-- optional nested specifications -->
    </analysisEngine>
            . . .
    <remoteAnalysisEngine key="key name">  <!-- 0 or more -->
      <casMultiplier poolSize="5"/>        <!-- req | omit-->
      <inputQueue ... />
      <replyQueue location="[local|remote]"/> <!-- optional -->
      <serializer method="xmi"/>              <!-- optional -->
      <asyncAggregateErrorConfiguration .../> <!-- optional -->
    </remoteAnalysisEngine>
            . . .
  </delegates>                . . .
</analysisEngine>
```

<analysisEngine> is used to specify deployment details for an analysis engine. It is optional, and if omitted, defaults will be used: The analysis engine will be run asynchronously, with a scaleout of 1, using the default error configuration.

The <scaleout ...> element specifies, for co-located primitive or non-AS aggregates (async="false") at the bottom of an aggregate tree, how many replicated instances are created.

The <casMultiplier> element inside an <analysisEngine> element is required if the analysis engine component is a CAS multiplier, and is an error if specified for other components. It specifies for CAS multipliers the size of the pool of CASes used by that CAS multiplier for generating extra CASes.

The <remoteAnalysisEngine> elements are used to specify that the delegate is not co-located, and how to connect to it. The <inputQueue> element specifies the remote's input queue. The <serializer> element describes what method of serialization to use (for now "xmi" is the only allowed value, and this element can be omitted). The casMultiplier poolSize inside a remoteAnalysisEngine element is only specified if the remote component is a CAS Multiplier, and it specifies the size of a pool of CASes kept to receive the new CASes from the remote component. Its size must be equal to or larger than the casMultiplier poolSize specified for that remote component.

> **Note:** Only one remote can be a remote CAS Multiplier, in the current design, and that remote can only be scaled "vertically" within one remote JVM; horizontal scaling (deploying multiple copies of the remote on different nodes, all servicing the same queue) is not supported in the current release

The <replyQueue> element specifies for delegates the location of the queue that receives replies from the delegate, for tcp: style connections. The two values allowed for location are "local" and "remote". Local means the reply queue is part of the process that is sending requests to the remote node; remote means the reply queue is on the same node as the remote process's input queue. The choice is dependent on both resource consumption (the queues store CASes in memory), and on firewall issues.

The default replyQueue location is local and normally does not have to be specified; users should set this to remote if a firewall prevents the remote delegate from accessing TCP/IP connections on the client's machine.

> **Note:** When replyQueue is set to remote, the brokerURL value used for this remote delegate must be valid for both the client to send requests and the remote service to send replies.

Services may be running on nodes with firewalls, where the only port open is the one for http. In this case, you can use the http protocol, For http: style connections, the only supported configuration is remote, and is the default.

The <asyncPrimitiveErrorConfiguration> element is only allowed within a top-level analysis engine specification (that is, one that is not a delegate of another, containing analysis engine).

3.8. Error Configuration descriptors

Error Configuration descriptors can be included directly in the deployment descriptors, or they may use the <import> mechanism to import another file having the specification.

For AS Aggregates, the configuration applicable to delegates goes in <asyncAggregateErrorConfiguration> elements for the delegate.

For AS Primitives, there is one <asyncPrimitiveErrorConfiguration> element that configures threshold-based termination. The other kinds of error configuration are not applicable for AS Primitives.

See Chapter 2, *Error Handling for Asynchronous Scaleout* [13] for a complete overview of error handling.

The Error Configuration descriptor for AS Aggregates is as follows; note that all the elements are optional:

```
<asyncAggregateErrorConfiguration
      xmlns="http://uima.apache.org/resourceSpecifier">

  <!-- the standard (optional) header -->
  <name>[String]</name>
  <description>[String]</description>
  <version>[String]</version>
  <vendor>[String]</vendor>

  <import ... />  <!-- optional -->

  <getMetadataErrors
          maxRetries="n"
          timeout="xxx_milliseconds"
          errorAction="disable|terminate"/>

  <processCasErrors
          maxRetries="n"
          timeout="xxx_milliseconds"
          continueOnRetryFailure="true|false"
          thresholdCount="xxx"
          thresholdWindow="yyy"
          thresholdAction="disable|terminate"/>

  <collectionProcessCompleteErrors
          timeout="xxx_milliseconds"
          additionalErrorAction="disable|terminate"/>

</asyncAggregateErrorConfiguration>
```

For an AS Primitive, the <asyncPrimitiveErrorConfiguration> element appears at the top level, and has this form:

```
<asyncPrimitiveErrorConfiguration
      xmlns="http://uima.apache.org/resourceSpecifier">

  <!-- the standard (optional) header -->
  <name>[String]</name>
  <description>[String]</description>
  <version>[String]</version>
  <vendor>[String]</vendor>

  <import ... />  <!-- optional -->

  <processCasErrors
          thresholdCount="xxx"
          thresholdWindow="yyy"
          thresholdAction="terminate"/>

  <collectionProcessCompleteErrors
          additionalErrorAction="terminate"/>

</errorConfiguration>
```

The maxRetries attribute specifies the maximum number of retries to do. If this is set to 0 (the default), no retries are done.

The continueOnRetryFailure attribute, if set to 'true' causes the framework to ask the aggregate's flow controller to attempt to continue, by skipping this failing delegate in its flow. The flow controller can indicate it cannot continue, in which case, an error result is returned.

> **Warning:** If maxRetries > 0 or the continueOnRetryFailure option is specified, the CAS will be saved before sending it to remote delegates, to enable the these actions. For co-located delegates, the CAS will *not* be copied; the retry or continue action is done with the CAS in the state where the failing delegate left it.

The timeout attribute specifies the timeout values used when sending the commands to the delegates.

The thresholdCount and thresholdWindow attributes specify a threshold, where xxx is the number of errors permitted in a window of size yyy. If the threshold is exceeded, the framework take the specified action of either disabling this delegate, or terminating the containing AS Aggregate (or if not an AS Aggregate, terminated the AS Primitive). Disabling removes the delegate from the flow, so the containing aggregate will no longer send it commands; the error result object returned indicates the disable action was taken. When disabling, the framework asks the flow controller to remove the delegate from the flow; the flow controller respond to the framework that it cannot reasonably operate without this component, in which case the framework will convert the "disable" into a "terminate".

The termination action disconnects the containing AS Aggregate or this AS Primitive from its input queue so it will no longer handle messages delivered to it. It cleans up the containing AS Aggregate or this AS Primitive, and, for AS Aggregates, cascades this cleanup down to any co-located delegates that are not shared by other clients.

If, as a result of termination, there are no more enabled consumers attached to a particular queue, the framework will set a flag and immediately return error results for both in-progress and future commands addressed to this queue. If, at some later point, one or more consumers become enabled on this queue, the framework will resume sending commands.

Note that the only action for an *AS Primitive* on getMetadata failure is to terminate, and this is always the case, so it is not listed as an configuration option. This is also the default and only action allowed if thresholds are used.

3.9. Error Configuration defaults

If the <errorConfiguration> element is omitted, or if some sub elements of this are omitted, the following defaults are used:

- Timeout defaults are set to 0 - meaning no timeout, except for the getMetadata command - here the default is 1 minute.
- No disable or terminate action will be done, except that the default for the getMetadata command is to terminate.

- No "continueOnRetryFailure" action is done.
- The maxRetries parameter is set to 0.